2023 年度 卒業論文

OCaml における Embedding by Unembedding

東北大学 工学部
電気情報物理工学科

C0TB2512 類家 健永

指導教員：住井 英二郎 教授
論文指導教員：松田 一孝 准教授

2024 年 2 月 28 日　14:00–14:30
青葉山キャンパス 電気系 3 号館 206 セミナー室

要旨

埋め込み領域特化言語は他の言語のライブラリの形で実装された言語であり、ホスト言語の機能やエコシステムを利用可能であるという利点がある。埋め込み領域特化言語の実装方式の一つにEmbedding by Unembedding (EbU) がある。EbU は双方向変換や漸増計算など複雑な意味の言語を、高階抽象構文を用いて実装することを可能とする。元の EbU は Haskell で実装されている。しかし、型クラス、GADT、型族、高階多相などの高度な機能が用いられているため、他の関数プログラミング言語での実現方式は明らかでなかった。本研究では OCaml における EbU の実現方式を示し、例を通してその有用性を評価する。

目次

第 1 章

序論

領域特化言語は特定の問題を解決するために、その問題の領域に特化した言語である。特に他の言語のライブラリの形で実装された領域特化言語は埋め込み領域特化言語と呼ばれる。埋め込み領域特化言語はその実装方式から、他の言語のユーザがホスト言語の機能やエコシステムを利用可能であるという利点がある。

埋め込み実装においては変数の取り扱いが問題となる。例えば、変数をそのまま文字列で表現するような素朴な実装は煩雑であり誤りやすい。解決策の一つに Higher-Order Abstract Syntax（HOAS）[5, 8, 10, 11] がある。HOAS はゲスト言語の変数束縛をホスト言語の関数で表現する。HOAS はユーザが利用しやすい一方、一部の言語については開いた式の意味をホスト言語の関数として表現することが難しいため HOAS の利用が自明ではない。例えば、双方向変換や増分計算、可逆計算がこれに該当する。

埋め込み領域特化言語の実装方式の一つに Embedding by Unembedding（EbU）[9] がある。EbU は開いた式の意味をホスト言語の関数として変換する `lift` 処理が提供されており、`lift` 処理により HOAS を用いて双方向変換や増分計算などを表現可能にする。元の EbU は Haskell で実装されている。しかし、EbU の現在の実装は Haskell の型クラス、GADT、型族、高階多相などの高度な機能が用いられているため、他の関数型プログラミング言語での実装方式は明らかではない。

本研究では、EbU をユーザビリティを維持しつつ実装するにはどのような言語機能が重要となるのかを明らかにするための最初のステップとして、他のメジャーな関数型プログラミング言語である OCaml で EbU の実装を行う。そして、増分計算の例により、その有用性を評価する。

本研究の貢献は以下である。

- EbU を他の関数型プログラミング言語で実装するときの問題点の特定（3.1 節）
- OCaml での新しい実装方式の提案（3.3 節）
- 増分計算の例により、OCaml 版 EbU でもユーザビリティが維持されていることを確認（4 章）

また、結論（6 章）を述べる前に、関連研究（5 章）について議論を行う。

第 2 章

準備

本章では、EbU を実装するにあたり、その基礎となる知識の準備を行う。EbU はユーザが扱う HOAS と意味領域上で動作する意味関数を接続するフレームワークとして機能する。したがって、本章では HOAS、Embedding by Unembedding について説明する。2.1 節では HOAS を、2.2 節では Embedding by Unembedding を扱う。

2.1 HOAS

Embedding by Unembedding（EbU）では Higher-Order Abstract Syntax（HOAS）を用いて言語を埋め込む。HOAS とは変数束縛の表現方法の一つであり、変数束縛をホスト言語の関数で表す。EbU で HOAS を用いる理由は、埋め込み言語の利便性の向上のためである。例えば、HOAS はゲスト言語の変数をホスト言語の関数で管理するため、言語実装者が変数の名前管理をする必要がない。

EbU では tagless-final style [4] により HOAS を埋め込む。tagless-final style は shallow embedding の一種である。tagless-final style では、ゲスト言語の式を多相型で表し、項および言語要素を抽象化し、解釈に応じて適切な実装を与える。

EbU は tagless-final style と unembedding [1, 2] を組み合わせたフレームワークであるため、tagless-final style は重要な技術となる。

2.2 Embedding by Unembedding

本研究は Haskell で実装される Embedding by Unembedding [9]（EbU）フレームワークを OCaml で実装することである。EbU は埋め込み領域特化言語の実装方法の一つであり、可逆計算、増分計算、双方向変換などの複雑な意味を持つ言語を統一されたインターフェースを用いて HOAS で埋め込むことを目的としている。この節では、EbU が対象とする言語の確認（2.2.1 節）や EbU の使い方（2.2.2 節）、その内部での処理（2.2.3 節）を紹介する。EbU の使い方では具体

例として、単純型付きラムダ計算（STLC）を扱う。

2.2.1 EbU の対象言語

はじめに EbU が対象とする言語から述べる。EbU は単純型付けが可能であり、一階か二階の言語要素 [7] を持ち、compositional な意味論を持ち、環境に対して weakening が適用できる言語を対象としている。ここで一階や二階とは言語要素の種類のことであり、二階は変数束縛を導入し、一階は変数束縛を導入しない。つまり、それぞれの言語要素は以下の形の型付け規則を持つ。

$$\frac{\{\Gamma \vdash e_i : A_i\}_{i=1,...,n}}{\Gamma \vdash con\ e_1\ ...\ e_n : B}$$

$$\frac{\{\Gamma, x_{i1} : A_{i1}, ..., x_{im_i} : A_{im_i} \vdash e_i : B_i\}_{i=1,...,n}}{\Gamma \vdash con'\ (x_{11}...x_{1m_1}.e_1)\ ...\ (x_{n1}...x_{nm_n}.e_n) : C}$$

2.2.2 EbU の使用例

続いて、EbU の使い方を紹介する。言語を埋め込むにははじめに、意味領域を特定する必要がある。これは埋め込む言語がどのようなふるまいをするかを考えることである。STLC では値環境を受け取り、結果を出力する意味を持つ。STLC の意味領域は Haskell のデータタイプとして以下のようにラップされる。

```
1 newtype STLC env a = Sim { runSim :: VEnv env -> a }
```

VEnv は値環境を表すが、具体的な実装方法はここでは述べない。埋め込む言語の意味領域を考えるとき、変数の型付け規則の解釈も準備する。変数の型付け規則の解釈には var と weaken の 2 種類が存在して、以下のように表される。

$$\frac{}{\Gamma, x : A \vdash x : A}\ \texttt{var}$$

$$\frac{\Gamma, x : A \vdash x : A \qquad y \notin dom(\Gamma)}{\Gamma, x : A, y : B \vdash x : A}\ \texttt{weaken}$$

つまり、var は項が必ず型付けできることを保証し、weaken は型付け環境を弱化できることを表す。Haskell では var と weaken は Variables 型クラスとして以下のようにカプセル化される。

```
1 class Variables (sem :: [k] -> k -> Type) where
2   var :: sem (a ': as) a
3   weaken :: sem as a → sem (b ': as) a
```

言語として埋め込む際には、var と weaken をインスタンス化し、解釈を与える必要がある。STLC では、var は値環境から最初の値を取り出し、weaken はの最初の値を無視して新しい値

環境を持つ Sim を作成する解釈を与える。Variables 型クラスは以下のようにインスタンス化される。

```
instance Variables STLC where
  var :: STLC (a ': as) a
  var = Sim (\(ECons (Identity x) _) → x)
  weaken :: STLC as a → STLC (b ': as) a
  weaken (Sim f) = Sim (\(ECons _ env) → f env)
```

次に埋め込む言語の変数以外の言語要素の意味を意味領域上の関数（意味関数）として与える。単純型付きラムダ計算には関数適用とラムダ抽象の二つの言語要素が存在する。そのため、関数適用を appSem、ラムダ抽象を lamSem として、意味関数をそれぞれ与える。appSem、lamSem の定義は以下になる。

```
appSem :: STLC env (a -> b) -> STLC env a -> STLC env b
appSem fTerm aTerm = Sim (\e ->
  let f = runSim fTerm e
      x = runSim aTerm e
  in f x)

lamSem :: STLC (a ': env) b -> STLC env (a -> b)
lamSem bTerm = Sim (\e x ->
  let e' = ECons (Identity x) e
  in runSim bTerm e')
```

続いて、埋め込み言語の構文を HOAS で表現する。意味関数は埋め込む言語の意味を正確に表現している反面、ユーザにとっては非常に使いにくい。そのため、EDSL のユーザが扱いやすいように言語の実装者は HOAS を与える必要がある。STLC の HOAS の構文は以下のように与える。

```
class STLChoas exp where
  lam :: (exp a -> exp b) -> exp (a -> b)
  app :: exp (a -> b) -> exp a -> exp b
```

以上により、埋め込む言語の HOAS と意味関数を定義できた。言語を動作させるには、意味関数から HOAS の構文に変換する意味モジュールを定義する必要がある。EbU にはこの接続に対して lift 関数を提供している。lift 関数により、高度な意味を持つ言語に対して、同様の操作で変換を行える。意味関数がどのような引数をとるかをあらかじめ指定する必要がある。app は一階の抽象構文であり、lam は二階の抽象構文である。意味関数から STCL の HOAS の構文への変換を以下に示す。

```
instance STLChoas (EnvI STLC) where
  app = UE.lift (LZ :. LZ :. End) appSem
  lam = UE.lift ((LS LZ) :. End) lamSem
```

ソースコード 2.1　HOAS の構文と意味関数の変換

最後に、言語を機能させる。open expression の解釈は `Variables` 型クラスの意味に対して、以下のように一般的に提供される。

```
runOpen :: Variables sem =>
           (EnvI sem a -> EnvI sem b) -> sem '[a] b
runOpen f = let eA = ECons Proxy ENil
                x  = EnvI $ \e -> weakenMany eA e var
            in runEnvI (f x) eA
```

`runOpen` は自由変数が一つである開いた式を受け取り semantic type に変換する。開いた式はホストレベルの関数として表現されるので、`runOpen` は開いた式を実行するための引数として、`EnvI sem` を作成する。STLC を実行する際、開いた式に含まれる自由変数が一つであることを確認するために、`runOpenSTLC` でラップする必要がある。以下に `runOpen` をラップするコードを示す。

```
runOpenSTLC :: (forall exp . STLChoas exp => exp a -> exp b)
                                           -> STLC '[a] b
runOpenSTLC f = runOpen f
```

最後に STLC を動作させるための関数を以下に示す。

```
runSTLC :: STLC '[a] b -> a -> b
runSTLC tx = let g = ECons (Identity x) ENil in runSim tg
```

開いた式を定義したら、`runSTLC(runOpenSTLC(open_term))` とすると、実行できる。
まとめると EbU による言語の実装手順は以下のようになる。

- step1. 意味領域を特定する。
- step2. 言語要素の意味関数を定義する。
- step3. 構文の HOAS 表現を提供する。
- step4. 構文に応じて、適切なリフティング関数を適用し、意味モジュールを定義する。
- step5. 言語を機能させる。

2.2.3　EbU の内部での処理

最後に、EbU の内部の処理について述べる。EbU で特に重要となる関数は `lift` 関数である。`lift` 関数は意味関数を HOAS に変換する関数であり、その変換の手順は以下のように表すことができる。

$(\forall env.Sem\ (A_{11}, ..., A_{1n} : env)\ A_1 \to ... \to Sem\ (A_{m1}, ..., A_{mk} : env)\ A_m \to Sem\ env\ A)$
↓

$(\forall env.Env\ (SemRep\ env)\ '[(A_{11}, ..., A_{1n} \rightsquigarrow A_1), ..., (A_{m1}, ..., A_{mk} \rightsquigarrow A_m)] \to Sem\ env\ A)$
$\downarrow$
$Env\ HoasRep\ '[(A_{11}, ..., A_{1n} \rightsquigarrow A_1), ..., (A_{m1}, ..., A_{mk} \rightsquigarrow A_m)] \to EnvI\ sem\ A$
$\downarrow$
$(EnvI\ sem\ A_{11} \to ... \to EnvI\ sem\ A_{1n} \to EnvI\ sem\ A_1) \to ... \to (EnvI\ sem\ A_{m1} \to ... \to EnI\ sem\ A_{mk} \to EnvI\ sem\ A_m) \to EnvI\ sem\ A$

EnvI sem は expression を表しており、A はその型を表す。SemRep と HoasRep に関しては後程説明する。lift 関数の大まかな流れは、項の型付け情報に対して、新しく環境に追加された型付け情報を集めて HOAS に変換することである。lift 関数の実装は複雑になる。複雑になる理由として、意味関数の各引数の形や数が異なるため，統一的な表現を経由しているためである。統一の操作を行うために、Sig2 というカインドを定義する。Sig2 の定義を以下に示す。

```
1 data Sig2 k = [k] :~> k
```

続いて、以下に定義されるデータ型を考える。

```
1 data SemRep (sem :: [k] -> k -> Type) (env :: [k])
2                                       (s :: Sig2 k) where
3   TR :: sem (Append as env) b -> SemRep sem env (as ':~> b)
4
5 data HoasRep (sem :: [k] -> k -> Type) (s :: Sig2 k) where
6   UR :: TEnv as -> (Env (EnvI sem) as -> EnvI sem b) ->
7                                          HoasRep sem (as ':~> b)
```

ソースコード 2.2　HoasRep と SemRep の定義

HoasRep は言語要素の引数の束縛変数の型のリスト as の値レベルの表現 TEnv as をとる。SemRep は入力された項の型付け規則を表現している。なお、Append は型レベルのリストの結合を表す関数で、以下のように定義される。

```
1 type family Append as bs where
2   Append '[] bs = bs
3   Append (a ': as) bs = a ': Append as bs
```

ソースコード 2.3　Append の定義

これらのデータ型を組み合わせることで、lift 関数のコアになる関数の定義が可能となる。この関数を liftCore とすると、その定義は以下のようになる。

```
1 liftCore :: forall sem ss r. Variables sem =>
2   (forall env. Env (SemRep sem env) ss -> sem env r)
3   -> Env (HoasRep sem) ss -> EnvI sem r
4 liftCore f ks = EnvI $ \e -> f (mapEnv (conv e) ks)
5   where conv :: TEnv env -> HoasRep sem s -> SemRep sem env s
```

```
6         conv e (UR e1 k) = TR $ cnv e e1 k
7         cnv :: TEnv env -> TEnv as ->
8                (Env (EnvI sem) as -> EnvI sem a) ->
9                sem (Append as env) a
10        cnv e e1 k = let {ex_e = appendEnv e1 e; xs = mkXs e e1 ex_e}
11                     in runEnvI (k xs) ex_e
12        mkXs :: TEnv env -> TEnv as' ->
13                 TEnv (Append as' env) -> Env (EnvI sem) as'
14        mkXs _ ENil _ = ENil
15        mkXs p (ECons _ as) te@(ECons _ te')
16          = let x = EnvI $ \e' -> weakenMany te e' var
17            in ECons x (mkXs p as te')
```

ソースコード 2.4　liftCore の定義

ここで、(Env (HoasRep sem) ss → EnvI sem r) の部分は埋め込まれていない解釈を表しており、(forall env. Env (SemRep sem env) ss → sem env r) は意味関数を表している。

liftCore 関数は言語の実装者が扱うには少々不便である。そのため、ソースコード 2.1 のように
lift ((LS LZ) :. End) lamSem のように意味関数とその引数のパラメータを渡すだけで、意味関数から HOAS の構文に変換する関数を用意する。はじめに、引数を表す環境を陰に扱うために、以下のような型族を定義する。

```
1  type family FuncSem (sem :: [k] -> k -> Type) (env :: [k])
2                      (ss :: [Sig2 k])
3                      (r :: k) | r -> sem env r where
4    FuncSem sem env '[] r = sem env r
5    FuncSem sem env ((as ':~> a) ': ss) r = sem (Append as env) a
6                                          -> FuncSem sem env ss r
7
8  type family FuncU (sem :: [k] -> k -> Type) (ss :: [Sig2 k])
9                    (r :: k) = res | res -> sem r where
10   FuncU sem '[] r = EnvI sem r
11   FuncU sem ((as ':~> a) ': ss) r = Func (EnvI sem) as (EnvI sem a)
12                                                    -> FuncU sem ss r
```

これらの型族を使って、以下のように lift 関数を定義できる。

```
1  lift :: forall sem ss r. Variables sem => Dim ss
2          -> (forall env. FuncSem sem env ss r) -> FuncU sem ss r
3  lift ns f =
4    let h :: forall env. Env (SemRep sem env) ss -> sem env r
5        h = fromF f
6    in toF ns (liftCore @sem h)
```

ソースコード 2.5　liftCoren の定義

`lift` 関数は `Dim ss` を受け取るが、`HoasRep` に必要な値レベルの表現を提供するために使用される。`lift` 関数は以下に示す型を持つ関数も使う。

```
1 toF :: Dim ss -> (Env (HoasRep sem) ss -> EnvI sem r) ->
2                                             FuncU sem ss r
3 fromF :: FuncSem sem env ss r ->
4                  Env (SemRep sem env) ss -> sem env
```

ソースコード 2.6　toF の定義

第 3 章

提案手法

3.1 EbU を OCaml で実装するときの問題点

Haskell 版の Embedding by Unembedding（EbU）では Haskell の高度な機能である型クラス、GADT、型族、高階多相が用いられる。しかし、OCaml では型クラス、型族、高階多相の機能は存在しない。特に EbU の実装に直接かかわる機能として、型族と高階多相があげられる。

型族は型レベルの関数を表現する。例えば、型レベルのリストの結合を定義したい場合、Haskell ではソースコード 2.3 のように定義する。結合したいリストに対して、再帰的に `Append` が適用され、リスト `as` と `bs` が結合されていることを確認できる。この型レベルのリストの結合を OCaml では直接実装することができない。

高階多相は型オペレータを抽象化できることであり、ソースコード 2.5 で導入されている `lift` の定義に用いられている。OCaml には高階多相を直接表現する機能が存在しないため、Haskell 版 EbU の `lift` 関数に対応する関数を表現することが難しい。

以上のように、Haskell の高度な機能を OCaml で記述できないことがある。このような高度な機能は別のデータ型を定義して回避したり、モジュールやファンクタを利用して書き換えたりすることで表現を行う。Haskell 版と同様に記述できる個所については、そのまま実装を行う。

3.2 OCaml による EbU

3.2 節では OCaml 版の Embedding by Unembedding（EbU）の使用例を示す。EbU の使用に関しては 2.3 節の手順に従って行う。例として、単純型付きラムダ計算（STLC）を扱う。

ステップ 1 として、実装する言語の意味領域を特定する。STLC の意味領域は、値環境を受け取り結果を返す。Haskell 版と同様に以下の OCaml の型として定義することで、OCaml のデータ型として実装する。STLC の意味領域の定義を以下に示す。

```
1 module VEnv = HList(struct type 'a t = 'a end)
2 type ('a, 'e) stlc = { runSim : 'e VEnv.hlist -> 'a }
```

環境はファンクタにより定義されており、異種混合リスト [6] を採用している。異種混合リスト

により、異なる型の値であっても、一つの環境で扱うことができる。以下のソースコードは異種混合リストの実装となる。

```
module type HLIST = sig
  type 'a el

  type _ hlist =
    | HNil : unit hlist
    | HCons : 'a el * 'r hlist -> ('a * 'r) hlist
end

module HList (E: sig type 'a t end)
      : HLIST with type 'a el = 'a E.t = struct ()
  type 'a el = 'a E.t

  type _ hlist =
    | HNil : unit hlist
    | HCons : 'a el * 'r hlist -> ('a * 'r) hlist
end
```

VEnv を定義することで変数の型付け規則の解釈である var と weaken を実装できる。Haskell では var と weaken を Variables 型クラスとして定義したが、OCaml ではファンクタにより定義する。OCaml では、ファンクタ Variables を定義する前に、モジュール型 VARIABLES を定義する必要がある。モジュール型 VARIABLES と、STLC におけるファンクタ Variables の定義を以下に示す。

```
module type VARIABLES = sig
  type (_, _) sem

  val var   : ('a, 'a * _) sem

  val weaken : ('a , 'r) sem -> ('a , _ * 'r) sem
end

module VarSTLC = Variables (
  struct
    type ('a, 'b) sem = ('a, 'b) stlc

    let var = { runSim = function VEnv.HCons (x, _) -> x }

    let weaken = fun stlc ->
      { runSim = function VEnv.HCons (_, venv') ->
                                        stlc.runSim venv' }
end )
```

ステップ 2 として、言語要素の意味関数を定義する。STLC には app と lam の二つの言語要素が存在する。
意味関数の定義を以下に示す。

```
module SemSTLC = struct
  open STLC

  let appSem = fun fTerm aTerm ->
    { runSim = fun venv ->
      let f = fTerm.runSim venv in
      let x = aTerm.runSim venv in
      f x }

  let lamSem = fun bTerm ->
    { runSim = fun venv x ->
      let g' = VEnv.HCons (x, venv) in
      bTerm.runSim g' }
end
```

step3 では、埋め込む言語の構文の HOAS 表現を提供する。HOAS での STLC の構文の提供を以下に示す。

```
module type HSTLC = sig
  type 'a expr

  val app : ('a -> 'b) expr -> 'a expr -> 'b expr
  val lam : ('a expr -> 'b expr) -> ('a -> 'b) expr
end
```

step4 では、STLC の意味関数を HOAS の構文に変換する意味モジュールを定義する。意味モジュールの定義では、言語要素がどのような引数を持つかを argSpec に記述する。argSpec はソースコード 2.1 の (LZ :. LZ :. End) や ((LS LZ) :. End) に対応する。app は引数の数が二つであり、一階の言語要素であるため、argSpec = XCons(XZ, XCons(XZ, XNil)) とする。lam は引数の数が一つであり、二階の言語要素であるため、argSpec = XCons(XS XZ, XNil) となる。lift 関数による意味モジュールの定義を以下に示す。

```
module STLC_sem : HSTLC with type 'a expr = 'a STLC.VarSTLC.envi =
  struct
    type 'a expr = 'a VarSTLC.envi
    open STLC open VarSTLC

    let app :
      type a b. (a -> b) envi -> a envi -> b envi = fun e1 e2 ->
      let argSpec = XCons (XZ, XCons (XZ, XNil)) in
        toF argSpec
```

```
10            (liftCore' {f = fun x -> fromF argSpec SemSTLC.appSem x})
11            e1 e2
12
13      let lam :
14        type a b. (a envi -> b envi) -> (a -> b) envi = fun e ->
15        let argSpec = XCons (XS XZ, XNil) in
16          toF argSpec
17            (liftCore' {f = fun x -> fromF argSpec SemSTLC.lamSem x})
18            e
19
20      let runOpenSTLC :
21        type a b.(a expr -> b expr) -> (b, a * unit) stlc = fun f ->
22          VarSTLC.runOpen f
23    end
```

ソースコード 3.1　意味モジュールの定義

ステップ 5 では、実際に言語を機能させる。EbU は Variables 型クラスの意味に対して一般的に開いた式の解釈を提供しており、OCaml 版でも同様に提供している。その解釈を以下に示す。

```
1  let runOpen : type a b. (a envi -> b envi) ->
2                                      (b, a * _) sem = fun f ->
3    let eA = TEnv.HCons ((), TEnv.HNil) in
4    let x  = { runEnvI = fun e' -> weakenMany eA e' H.var} in
5    (f x).runEnvI eA
```

自由変数が一つであるケースには、runOpen は開いた式を受け取り、意味領域に変換する。自由変数が一つであることを確認するには、runOpenSTLC でラップする必要がある。以下に runOpenSTLC を示す。

```
1  module type OPEN_STLC_TERM = sig
2    type in_t
3    type out_t
4
5    module OpenTerm(L : HSTLC) : sig val res : in_t L.expr ->
6                                                out_t L.expr end
7  end
8
9  type ('a, 'b) open_stlc_term =
10   (module (OPEN_STLC_TERM with type in_t = 'a and type out_t = 'b))
11
12 let runOpenSTLC : type a b.
13   (a, b) open_stlc_term -> (b, a * unit) stlc =
14     fun (module OT) ->
15       let module M = OT.OpenTerm(STLC_sem) in
16       STLC_sem.runOpenSTLC M.res
```

ここまでで、一般的な開いた式に対する実行関数を定義したため、次に STLC を動作させる関数を定義する。STLC を動作には、以下の関数を利用する。

```
let runSTLC = fun t x ->
  let g = VEnv.HCons (x, VEnv.HNil) in
  t.runSim g
```

この関数は自由変数が一つの STLC 項に対して動作する。実際に、値 3 を持つ自由変数への恒等関数の適用を評価する。すると、以下のソースコードの実行に対して、次のような結果が得られる。

```
module TM = struct
  type in_t = int type out_t = int
  module OpenTerm(L : HSTLC) = struct
    open L
    let res x = app (lam (fun y -> y)) x
  end
end

let ex = fun x ->
  STLC_sem.runSTLC (STLC.runOpenSTLC (module TM)) x

ex 3
```

```
val ex : 'a STLC.STLC_sem.expr → 'a STLC.STLC_sem.expr = <fun>
val ex : int = 3
```

3.3 liftCore の実装

lift 関数は Haskell の高度な機能を使い実装されている。OCaml で実装は straightforward ではないが、Haskell 版の EbU と同様の手順で実装する。　はじめに、sig2 データ型を用意する。sig2 の定義は以下のように示す。

```
type ('xs, 'x) sig2 = |
```

続いて、HoasRep と SemRep のデータ型を定義する。この定義を以下に示す。

```
type (_,_) SemRep =
  | TR : ('xs,'env,'r) wapp * ('x, 'r) sem ->
                    ('env, ('xs, 'x) sig2) SemRep

type (_) HoasRep =
  | UR : 'xs TEnv.hlist * ('xs HListEnvI.hlist -> 'x envi) ->
                                          ('xs,'x) sig2 HoasRep
```

ここで、ソースコード 2.2 に注目する。SemRep の定義には、型環境の対する Append が含まれており、この Append は型族を用いて定義されている。しかし、OCaml では型族を表すことができない。そのため、Haskell の Append に該当する操作として、新しく wapp を定義する。wapp の定義を以下に示す。

```
type (_,_,_) wapp =
  | AppNil  : (unit, 'b, 'b) wapp
  | AppStep : ('x , 'y , 'r) wapp -> ('a * 'x, 'y, 'a * 'r) wapp
```

wapp は型レベルのリストが入れ子の組で表現されている。wapp は三つ型パラメータを持ち、その関係は ('x, 'y, 'r) wapp に対して
$'r = 'x +\!+ 'y$ となる。これにより、疑似的に Append を表現でき、OCaml で直接表現できない型の表現を回避できる。

以上により、liftCore 関数の実装の準備ができた。以下に liftCore 関数の実装を示す。

```
let liftCore :
  type ss rr. (ss,rr) semTerm -> ss HoasRepEnv.hlist -> rr envi =
    fun ff ks ->
    { runEnvI = fun (type env) (e : env TEnv.hlist) ->
      let module App = HListP(TEnv) in
      let module M_map = MapToSemRepHList(HoasRepEnv) in
      let rec mkXs : type env ys ys_env. env TEnv.hlist
          -> ys TEnv.hlist
          -> (ys, env, ys_env) wapp
          -> ys_env TEnv.hlist -> ys HListEnvI.hlist =
           fun p ys wit te ->
           match ys, wit, te with
           | HNil, _, _ -> HNil
           | (HCons(_,ys')),AppStep(wit'),HCons(_,te') ->
             let x = { runEnvI = fun e' -> weakenMany te e' H.var}
              in HCons(x,mkXs p ys' wit' te')
          in
      let cnv : type env xs x. env TEnv.hlist -> xs TEnv.hlist
                  -> (xs HListEnvI.hlist -> x envi)
                  -> (env, (xs,x) sig2) SemRep = fun e e1 k ->
         match App.append_hlist e1 e with
         | AppHList(wit, ex_e) ->
           let xs = mkXs e e1 wit ex_e in
           TR(wit, (k xs).runEnvI ex_e) in
      let conv : type env s. env TEnv.hlist -> s HoasRep ->
                                               (env,s) SemRep =
       fun e ur -> match ur with
       | UR(e1, k) -> cnv e e1 k in
      ff.f (M_map.map {f = fun xs -> conv e xs} ks)
    }
```

liftCore 関数の型に注目する。Haskell 版の liftCore 関数の型は (forall env. Env (SemRep sem env) ss → sem env r) → Env (HoasRep sem) ss → EnvI sem r となっているが、OCaml 版の型では (ss,rr) semTerm → ss HoasRepEnv.hlist → rr envi となっており、第一引数の型が一見異なるように見える。これは OCaml では高ランク多相を直接表現できないためであり、代わりにフィールドに多相型のレコードを用意する。semTerm の定義を以下に示す。

```
1 type ('ss,'rr) semTerm = {f: 'env. ('env,'ss) SemRepEnv_hlist ->
2                                                    ('rr, 'env) sem }
```

SemRepEnv_hlist は Haskell 版における Env (SemRep sem env) ss に対応しており、以下のように定義される。

```
1 type ('env,_) SemRepEnv_hlist =
2   | THNil  : ('env,unit) SemRepEnv_hlist
3   | THCons : ('env,'s) SemRep * ('env, 'ss) SemRepEnv_hlist
4                               -> ('env,'s * 'ss) SemRepEnv_hlist
```

Haskell 版の定義では Env (SemRep sem env) ss は高階多相を利用して定義されているが、3.1 節でも述べた通り OCaml には高階多相を直接表現する手段がない。そのため、GADT を利用することで、高階多相が必要になることを回避した。以上を踏まえ、semTerm の定義を見ると、(forall env. Env (SemRep sem env) ss → sem env r) を表せることが確認できる。

3.4 利便性の高い API の提供

HOAS の構文と意味関数の変換を行う関数は liftCore 関数であるが、Haskell 版の lift 関数を見ると、toF や fromF といった、補助関数が見受けられる。これらの関数はユーザの利便性を向上させるために導入された関数である。そのため、本節でもユーザビリティのため、toF や fromF を実装する。

はじめに、toF 関数を定義する。toF 関数は環境に追加された型情報を HOAS に変換する関数である。しかし、関数の内容自体は複雑ではなく、アンカリー化をするような動作を行う。そこであらかじめ、アンカリー化を行うファンクタを以下に定義する。

```
1 module CurryUncurry(H : HLIST) = struct
2   type (_,_,_) cspec =
3     | Z : ('r, 'r, unit) cspec
4     | S : ('r, 'f, 'xs) cspec ->
5           ('r, 'a H.el -> 'f, ('a * 'xs)) cspec
6
7   let rec uncurry :
```

```
8       type r f xs. (r,f,xs) cspec -> f -> xs H.hlist -> r =
9       fun s h args -> match s, args with
10      | Z ,   _            -> h
11      | S ss, HCons(x,xs) -> uncurry ss (h x) xs
12
13    let rec cspec2TEnv :
14      type r f xs. (r, f, xs) cspec -> xs TEnv.hlist =
15        fun n -> match n with
16        | Z -> TEnv.HNil
17        | S s' -> TEnv.HCons ((), cspec2TEnv s')
18  end
```

cspec 型は uncurry を行う回数を表すデータ型である。cspec2TEnv は cspec と同じ大きさの型リストを作成する関数である。続いて、toF の型を考える。ソースコード 2.6 を見ると、toF 関数の定義に型族が用いられている。OCaml では型族を直接表現できないため、FuncU に対応する GADT を以下にに定義する。

```
1  type (_,_,_) tf_spec =
2    | SNil  : ('rr, 'rr, unit) tf_spec
3    | SCons : ('x envi, 'f, 'xs) cspec * ('rr, 'ff, 'ss) tf_spec ->
4              ('rr, 'f -> 'ff, (('xs, 'x) sig2 * 'ss)) tf_spec
```

tf_spec は FuncU のほかに、Dim にも対応している。これにより、toF 関数の定義の準備ができた。toF 関数の定義を以下のように示す。

```
1  let toHoasRep : type f xs x. (x envi, f, xs) cspec ->
2    f -> (xs,x) sig2 HoasRep
3      = fun cs f -> UR (cspec2TEnv cs, uncurry cs f)
4
5  let rec toF : type rr ff ss. (rr envi,ff,ss) tf_spec ->
6    (ss HoasRepEnv.hlist -> rr envi) -> ff
7      = fun tfs h -> match tfs with
8        | SNil -> h HNil
9        | SCons(cs,tfs) -> fun k ->
10         toF tfs (fun r -> h (HCons(toHoasRep cs k,r)))
```

続いて、fromF の定義を行う。fromF 関数は型付け環境に新しく追加された型付け規則を集める関数である。fromF 関数の実装は toF 関数の実装方法と同じである。fromF 関数も toF 関数と同様にアンカリー化をする関数であるが、型付け規則を収集するため、それに対応する新しい GADTFuncSem を導入する必要がある。以下にに FuncSem に対応する GADT を示す。

```
1  type (_,_,_,_,_,_) ff_spec =
2    | FNil  : ('dummy, 'env,'rr, ('rr,'env) sem,unit) ff_spec
3    | FCons : ('xs,'env,'app) wapp  *
4              ('dummy,'env,'rr,'ff,'ss) ff_spec ->
```

```
5              ('app * 'dummy, 'env, 'rr, ('x,'app) sem ->
6                'ff , (('xs,'x) sig2 * 'ss)) ff_spec
```

ff_spec には dummy 型が含まれる。この dummy 型は app 型を保存するために利用される。本来、app 型は型推論により型を決定されるが、OCaml では app 型の型推論を行うことができない。そのため、意図的に dummy 型を導入して、app 型の型を推論できるようにする。

ff_spec により fromF 関数が定義できる。以下にに fromF 関数を定義する。

```
1  let rec fromF :
2    type dummy env res tt ss.
3      (dummy, env, res, tt, ss) ff_spec -> tt ->
4      ((env, ss) SemRepEnv_hlist -> (res,env) sem) =
5        function
6        | FNil -> fun x _ -> x
7        | FCons (w1, fs) -> fun f -> function
8          | THCons(TR(w2,t),ts) -> match wapp_functional w1 w2 with
9            | Refl -> fromFunc fs (f t) ts
```

toF 関数と同様、型族の実装以外は fromF 関数の定義と異なる点はない。

以上により、lift 関数の定義の準備が整った。lift 関数の利用方法はソースコード 3.1 に示すとおりである。toF 関数と fromF 関数で扱う GADT は型は異なるが、形は同じになる。EbU の利便性のため、toF 関数と fromF 関数で扱う GADT はまとめて定義したい。そこで、新しい GADT として以下の GADT を定義する。

```
1  type (_,_,_,_,_,_) arg_spec =
2    | XNil  : ('dummy, 'env, 'rr, 'rr envi,
3               ('rr,'env) sem, unit) arg_spec
4    | XCons :
5        ('xs, 'env, 'app, 'x envi, 'f) wapp_cspec *
6        ('dummy, 'env, 'rr, 'toFunc, 'fromFunc, 'ss) arg_spec ->
7        ('app * 'dummy, 'env, 'rr, 'f -> 'toFunc, ('x,'app) sem ->
8                        'fromFunc, (('xs,'x) sig2 * 'ss)) arg_spec
```

ソースコード 3.2　arg_spec の定義

この GADT により、ユーザが記述する言語要素の引数の情報は一つだけでよくなる。ソースコード 3.1 で利用される toF や fromF は、arg_spec を tf_spec や ff_spec に変換し、toF 関数や fromF 関数に適用する関数である。

以上より、OCaml 版 EbU の lift 関数を定義できる。しかし、OCaml 版 EbU と Haskell 版 EbU の lift 関数の記述方法を比較すると少々相違点が見受けられる。Haskell 版の EbU の lift 関数は言語要素の引数の情報と意味関数を引数として与えるだけで、HOAS と意味関数の変換を行える。一方で、OCaml 版の EbU の lift 関数は argSpec で言語要素の各引数の束縛の情報を表現し、toF、fromF、liftCore、argSpec の 4 つの関数を利用することで意味モジュールの定義を行う。OCaml 版の EbU は、本来言語実装者に見せたくない toF や fromF、liftCore を明示的に

与えている。この原因も 3.1 節で述べた OCaml が高階多相を直接表現する機能がないことに起因する。実際に、以下の素朴な `lift` 関数の実装は型整合ではない。

```
1 let lift = fun e argSpec sem ->
2   toF argSpec (liftCore' {f = fun x -> fromF argSpec sem x}) e
```

仮に OCaml に高階型変数を表現する機能が存在すれば、Haskell 版の `lift` 関数の型 `forall sem ss r. Variables sem => Dim ss -> (forall env. FuncSem sem env ss r) -> FuncU sem ss r` を、簡便のため一階の高ランク多相を許したとして`'env. (...,'env,'r,'t,'env,'f,'ss) arg_spec -> 'env. 'env 'f -> 't` と書ける。しかし、'f が高階の型変数であるためこのような型は OCaml で表現可能ではない。ファンクタを用いることで同等の型が表現できることが予想されるが、素朴にはソースコード 3.2 でいうところの'x や'xs を陽に渡す必要が生じ、ユーザビリティを大きく損う。

この問題を解決するために ppx を利用することを検討している。ppx は OCaml で利用できるプリプロセッサである。ppx により、`toF`、`fromF`、`liftCore` を隠蔽し、Haskell 版の `lift` 関数に対応する関数を OCaml で提供することが可能となる。`lift` 関数を OCaml で提供できれば、ユーザビリティの高い API を EbU ユーザに提供可能できる。
もしくは、5 章で述べる Yallop と White の手法 [14] を利用することで functor に頼らない実装が可能になると考えれらる。これに関して詳しくは、5.2 節にて述べる。

第 4 章

評価

4.1 EDSL の実装方法

3 章で定義した Embedding by Unembedding（EbU）を用いて、増分計算（ILC）[3] の埋め込みを行う。ILC はユーザ入力された計算と、その差分の計算を同時に行う計算体系である。例として、入力された自然数に対して 2 乗を行う式を考える。ユーザが

$f\ x\ =\ x^2$

と入力すると、システム内部ではユーザ入力に対する差分として、

$df\ x\ dx\ =\ 2\ *\ x\ *\ dx\ +\ dx\ *\ dx$

を計算する。

結果を得るには入力した式に対して、具体的な値を代入すればよい。例えば、通常の結果を得たいとき、ユーザの入力 $f\ 5$ に対して、25 を出力する。差分の結果が得たいとき、$x\ =\ 5$ に対して、その差分 2 の結果は $df\ 5\ 2$ を入力すると、24 が得られる。これは $f\ (5\ +\ 2)\ -\ f\ 5$ の結果と一致する。

ILC の意味領域は一般的に以下のようになる。

$A \to B,\ A \to \Delta A \to \Delta B$

A は入力、ΔA は入力の差分、B は出力、ΔB は出力の差分を表す。つまり、ILC は以下の二つの意味領域を持つ。

- 通常の評価を行い、入力に対して結果を出力する
- 差分の評価を行い、入力と入力の差分に対して結果の差分を出力する

この意味領域を、二つのアイディアによって定義する。

一つ目は対象言語の項に対する型付けを $e\ :\ A$ から $e\ :\ <A,\ \Delta A>$ に拡張して、A と ΔA の両方行うことである。これは、あくまで型付けを二つ行うことであり、組として扱うわけではない。

二つ目は、$e\ :\ <A,\ \Delta A>$ を組として表現して、

$[(A_1, \Delta A_1), ..., (A_n, \Delta A_n)]\ H.hlist \simeq A_1 * ... * A_n * unit$

として、型付け環境を表すことである。hlist とすることで、型環境の表現を容易に行うことができる。ただし、この表現方法のみだと map や concat 等の実装が困難になる。一般には A と ΔA のだけではなく、項の変更を反映するための関数も保持する必要がある。この問題は EbU の問題ではなく、ILC の複雑さに起因するものである。そのため、本研究ではこの問題を扱わず、項を反映するための関数の保持が必要のないケースについてのみ議論する。

以上のことをふまえ、意味領域は以下のようになる。

```
type ('x, 'xs) ilc =
    ('xs H.hlist -> 'x FromFst.t)
  * ('xs H.hlist -> 'xs D.hlist -> 'x FromSnd.t)
```

これは $\Gamma \vdash e : A$ の意味領域 $(\llbracket\Gamma\rrbracket \to \llbracket A\rrbracket) \times (\llbracket\Gamma\rrbracket \to \Delta\llbracket\Gamma\rrbracket \to \Delta\llbracket A\rrbracket)$ に対応する。FromFst や FromSnd は ILC の出力である B や ΔB を表すモジュールである。

意味領域を定義できたため、残りの埋め込みに関しては 3 章 2 のとおりに行う。まず、ILC の意味関数は加算を行う addSem、乗算を行う mulSem、組の第一要素を取る fstSem、組の第一要素を取る sndSem、変数を束縛する letSem を定義する。以下にに ILC の意味関数の定義を示す。

```
module SemILC = struct
open ILC

let addSem : type env.
  (int * int, env) ilc -> (int * int, env) ilc ->
  (int * int, env) ilc =
    fun (f, df) (g, dg) ->
      (fun theta ->
        let FFst v = f theta in
        let FFst w = g theta in
        FFst (v + w)) ,
      (fun theta dtheta ->
        let FSnd dv = df theta dtheta in
        let FSnd dw = dg theta dtheta in
        FSnd (dv + dw))

let mulSem : type env.
  (int * int, env) ilc -> (int * int, env) ilc ->
  (int * int, env) ilc =
    fun (f, df) (g, dg) ->
      (fun theta ->
        let FFst v = f theta in
        let FFst w = g theta in
        FFst (v * w)),
      (fun theta dtheta ->
        let FFst v = f theta in
        let FFst w = g theta in
```

```
28          let FSnd dv = df theta dtheta in
29          let FSnd dw = dg theta dtheta in
30          FSnd (v * dw + dv * w + dv * dw)
31          )
32
33  let fstSem : type env a da b db.
34    ((a * b) * (da * db) , env) ilc -> (a * da, env) ilc =
35      fun (f, df) ->
36        (fun theta ->
37         let FFst (v, _) = f theta in
38             FFst v),
39        (fun theta dtheta ->
40         let FSnd (dv,_) = df theta dtheta in
41             FSnd dv
42        )
43
44  let sndSem : type env a da b db.
45    ((a * b) * (da * db) , env) ilc -> (b * db, env) ilc = ...
46
47  let letSem : type env a b.
48    (a, env) ilc -> (b, (a * env)) ilc -> (b, env) ilc =
49      fun (f, df) (g, dg) ->
50        (fun theta ->
51         let FFst v = f theta in
52         let FFst w = g (HCons(FFst v,theta)) in
53             FFst w),
54        (fun theta dtheta ->
55         let FFst v = f theta in
56         let FSnd dv = df theta dtheta in
57             dg (HCons(FFst v, theta)) (HCons(FSnd dv,dtheta))
58        )
59  end
```

続いて、ILC の HOAS を与える。HOAS は let_、add、mul、fst、snd の 5 つを定義する。ILC の HOAS を以下に示す。

```
1  module type HILC = sig
2    type 'a expr
3
4    val let_ : 'a expr -> ('a expr -> 'b expr) -> 'b expr
5    val add  : (int * int) expr -> (int * int) expr -> (int * int) expr
6    val mul  : (int * int) expr -> (int * int) expr -> (int * int) expr
7    val fst  : (('a * 'b) * ('da * 'db)) expr -> ('a * 'da) expr
8    val snd  : (('a * 'b) * ('da * 'db)) expr -> ('b * 'db) expr
9  end
```

続いて、意味関数と HOAS を接続する。let_のみ second の言語要素であるため、lift 関数に渡す引数の情報について注意する。以下にに意味関数と HOAS を接続するプログラムを示す。

```
module HILC_sem : HILC = struct
  open ILC
  type 'a expr = 'a VarILC.envi
  open VarILC

  let let_ = fun e1 e2 ->
    let argSpec = XCons (Z, XCons (S Z, XNil)) in
    toF argSpec
    (liftCore' {f = fun x -> fromF argSpec SemILC.letSem x}) e1 e2
    e1 e2

  let add = fun e1 e2 ->
    let argSpec = XCons (Z, XCons (Z, XNil)) in
    toF argSpec
    (liftCore' {f = fun x -> fromF argSpec SemILC.addSem x}) e1 e2

  let mul = fun e1 e2 ->
    let argSpec = XCons (Z, XCons (Z, XNil)) in
    toF argSpec
    (liftCore' {f = fun x -> fromF argSpec SemILC.mulSem x}) e1 e2

  let fst = fun e ->
    let argSpec = XCons (Z, XNil) in
    toF argSpec
    (liftCore' {f = fun x -> fromF argSpec SemILC.fstSem x}) e

  let snd = fun e ->
    let argSpec = XCons (Z, XNil) in
    toF argSpec
    (liftCore' {f = fun x -> fromF argSpec SemILC.sndSem x}) e
end
```

これにより、ILC を埋め込むことができたため、あとは ILC を実行すればよい。

4.2 Haskell 版との比較

OCaml 版の EbU と Haskell 版の EbU で実装された ILC には比較すると大きく二つの相違点がある。

一つ目は意味領域の定義方法である。Haskell 版では ILC の意味領域を以下のように定義している。

```
data ILC env a = Inc
```

```
2   { sEval :: VEnv env -> a,
3     dEval :: VEnv env -> DEnv env -> Delta a }
```

ソースコード 4.1　Haskell における ILC の意味領域

Haskell 版の ILC と OCaml 版の ILC では意味領域の考え方は同じであるが、その定義方法が大きく異なる。Haskell 版では `DEnv` の定義を open type family によって行う。これにより、`DEnv` の型を後から変更することができるため、項によらずに型環境を扱うことができる。OCaml では型族の代わりに GADT を用いて実装を行ったが、OCaml の GADT ではあらかじめ型を記述する必要があるため、open type family を GADT で表現できない。そのため、OCaml ではまず、通常の入力と差分の入力を表すモジュールを以下のように定義した。

```
1  module FromFst = struct
2    type _ t =
3    | FFst : 'a -> ('a * 'd) t
4  end
5
6  module FromSnd = struct
7    type _ t =
8    | FSnd : 'd -> ('a * 'd) t
9  end
```

次に、`FromFst` や `FromSnd` を要素とする `hlist` を考え、`DEnv` と同等の表現力を持たせた。本研究で扱かった ILC はこのような方法で意味領域を表現することができたが、扱う計算体系によっては意味領域を OCaml で表現しきれない可能性がある。本論文では OCaml 版の EbU で埋め込むことができる計算体系の範囲は議論しないが、これは今後検討したい。

二つ目は `lift` 関数の利用方法である。ただし、この議論は 3.4 でも行ったため、省略する。

以上の 2 点が OCaml 版の EbU と Haskell 版の EbU の相違点である。ILC 以外の高度な意味の意味領域の実装方法、OCaml における高階多相のフィールドやファンクタを用いない表現方法が明らかになれば OCaml 版の EbU も Haskell 版の EbU と同等のユーザビリティが得られると考えられる。

第 5 章

関連研究

5.1 typeindexed value の表現

本研究では、toF や fromF を GADT arg_spec を引数に取る関数として実装したが、これらは Yang [15] の type-indexed value の表現手法の一つを用いても実装可能でなる。type-indexed value を GADT を用いて、f :: Sing a -> F a のような形で値レベルの表現を表す GADT を引数に取ることで表せる。実際に fromF で扱う GADT（ff_spec）を、Yang の手法で表すと以下のようになる。

```
let fnil : ('r,'env) sem -> ('env,unit) SemRepEnv_hlist
                         -> ('r,'env) sem = fun x _ -> x
let fcons :
  ('xs,'env,'app) wapp
  -> ('ff -> ('env,'ss) SemRepEnv_hlist -> ('r,'env) sem)
  -> ((('x,'app) sem -> 'ff)
  -> ('env, ('xs,'x) sig2 * 'ss) SemRepEnv_hlist
  -> ('r,'env) sem)
  = fun _ h f -> function
      | THCons(TR(w2,t),ts) -> h (f (Obj.magic t)) ts
```

ソースコード 5.1　ff_spec を表す type-indexed value

この手法の利点の 1 つとして、arg_spec や ff_spec で出現する'dummy を考える必要がない点である。実際に、ソースコード 5.1 を見ると、'dummy に相当する箇所がないことがわかる。一方で、SemRepEnv_hlist のようなユーザには見せる必要のない実装の詳細が陽に現れるというデメリットもある。

5.2 ファンクタに頼らない高階多相の実装

通常、OCaml では高階多相の抽象化にはファンクタを用いる必要がある。しかし、Yallop と White はファンクタを用いずに、defunctionalization を型レベルで行うことで高階多相を表現す

る [14]。defunctionalization とは、λ 抽象をその自由変数の値を引数にとるようなコンストラクタで表すことで、関数抽象や関数適用を一階の言語で表す方法である [12, 13]。

Yallop と White のアイデアは型オペレータ op 毎にその型オペレータを表す型コンストラクタ op_rep を用意し、また、エンコードされた型オペレータの適用を表す型 (_,_) app を ('a, op_rep) app と'a op が同型になるように定義する。これにより、'f. ...t 'f... のような高階多相を'f_rep. ...(t, 'f_rep) app... という一階の多相で表現できるようになる。

この手法により、3.4 節で述べた lift 関数の問題を解決できることが予想される。そのためには'env に依存した型を'env を受けとるような型オペレータとして陽に表現する必要があり、arg_spec 型のみならず wapp 型の定義やそれを利用する関数など広範な変更が必要となることが予想される。

第 6 章

結論

本稿では Embedding by Unembedding [9] と呼ばれる複雑な意味を持つ言語の埋め込み手法の実装に必要な機能を明らかにすることが目的であった。Embedding by Unembedding は Haskell で実装されており、その実装には Haskell の高度な機能である型クラス、型族、高階多相が用いられる。そのため、他の関数型プログラミング言語では、Haskell 版 EbU のユーザビリティを保持したまま実装する方法が明らかではない。本研究では型族や高階多相を GADT で表現することで、ユーザビリティを保ったまま EbU を実装した。したがって、EbU の実装には必ずしも型族や高階多相が必要でないが、GADT が実装に必要であることがわかった。また、作成した EbU フレームワークを利用して、複雑な埋め込みの例である増分計算を埋め込むことに成功した。ただし、完全な EbU の実装はできておらず、増分計算以外の高度な意味を持つ言語の埋め込みを確認できていない。今後は EbU の完全な実装と、他の複雑な意味を持つ言語の埋め込みを検討している。

謝辞

本研究を進めるにあたり、論文指導教員である松田一孝准教授に様々なご指導を賜りました。深く感謝申し上げます。また、様々な面で篤いご指導とご厚意を与えてくださり、大変お世話になった指導教員の住井英二郎教授、並びに同学科松田一孝准教授、並びに同学科 Oleg Kiselyov 助教に深く御礼申し上げます。また、住井・松田研究室の皆様には研究するにあたり、多くの激励をいただきました。ここに深謝の意を表し謝辞といたします。

参考文献

[1] Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, Vol. 5608 of *Lecture Notes in Computer Science*, pp. 35–49. Springer, 2009.

[2] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pp. 37–48. ACM, 2009.

[3] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pp. 145–155. ACM, 2014.

[4] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, Vol. 19, No. 5, pp. 509–543, 2009.

[5] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, Vol. 5, No. 2, pp. 56–68, 1940.

[6] Kiselyov Oleg Evgenievich. Typed heterogeneous collections: Lightweight hlist, 2018. https://okmij.org/ftp/ML/index.html.

[7] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pp. 193–202. IEEE Computer Society, 1999.

[8] Gérard P. Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, Vol. 11, pp. 31–55, 1978.

[9] Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. Embedding by unembedding. *Proc. ACM Program. Lang.*, Vol. 7, No. ICFP, pp. 1–47, 2023.

[10] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating

formulas and programs. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987*, pp. 379–388. IEEE-CS, 1987.

[11] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pp. 199–208. ACM, 1988.

[12] John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pp. 717–740. ACM, 1972.

[13] John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, Vol. 11, No. 4, pp. 363–397, 1998.

[14] Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, Vol. 8475 of *Lecture Notes in Computer Science*, pp. 119–135. Springer, 2014.

[15] Zhe Yang. Encoding types in ml-like languages. *Theor. Comput. Sci.*, Vol. 315, No. 1, pp. 151–190, 2004.

付録

付録では実装した OCaml 版 EbU のコードを載せる。ソースコード 1 では EbU の実装を、ソースコード 2 には 3 章で埋め込んだ単純型付きラムダ計算のコードを、ソースコード 3 には 4 章で埋め込んだ増分計算のコードを示す。

```
1  module type HLIST = sig
2    type 'a el
3    type _ hlist =
4      | HNil : unit hlist
5      | HCons : 'a el * 'r hlist -> ('a * 'r) hlist
6
7    val length : 'r hlist -> int
8  end
9
10 module HList (E: sig type 'a t end) : HLIST
11                           with type 'a el = 'a E.t = struct ()
12   type 'a el = 'a E.t
13   type _ hlist =
14     | HNil : unit hlist
15     | HCons : 'a el * 'r hlist -> ('a * 'r) hlist
16
17   let rec length : type r. r hlist -> int = function
18       HNil        -> 0
19     | HCons(_,r) -> 1 + length r
20 end
21
22 module TEnv = HList(struct type 'a t = unit end)
23
24 module HL = HList( struct type 'a t = 'a end )
25
26 type (_,_,_) wapp =
27   | AppNil  : (unit, 'b, 'b) wapp
28   | AppStep : ('x , 'y , 'r) wapp -> ('a * 'x, 'y, 'a * 'r) wapp
29
30 module type TypeM = sig
31   type t
```

```
end

type (_,_) equal =
  | Refl : ('a,'a) equal

let rec wapp_functional :
  type xs ys zs1 zs2.
  (xs,ys,zs1) wapp -> (xs,ys,zs2) wapp -> (zs1,zs2) equal = function
  | AppNil -> begin function AppNil -> Refl end
  | AppStep w1 -> function AppStep w2 ->
    match wapp_functional w1 w2 with
    | Refl -> Refl

module HListP(H: HLIST) = struct
  include H

  type (_,_) app_hlist =
  | AppHList : ('x,'y,'r) wapp * 'r hlist -> ('x ,'y) app_hlist

  let rec append_hlist : type x y. x hlist -> y hlist
                                   -> (x,y) app_hlist
    = fun xs ys ->
      match xs with
      | HNil -> AppHList (AppNil, ys)
      | HCons(a,r) ->
        match append_hlist r ys with
        | AppHList (w,res) -> AppHList(AppStep w, HCons(a,res))
end


module type VARIABLES = sig
  type (_, _) sem

  val var   : ('a, 'a * _) sem
  val weaken : ('a , 'r) sem -> ('a , _ * 'r) sem
end

module Variables(H: VARIABLES) = struct
  include H

  let rec go : type a a' b. int -> a TEnv.hlist -> a' TEnv.hlist
                                       -> (b, a) sem -> (b, a') sem
    = fun n a b sem ->
      match n with
      | 0 -> Obj.magic sem
```

```
77      | _ -> match b with
78             | TEnv.HNil -> failwith "Cannot happen"
79             | TEnv.HCons (_, b') -> weaken (go (n-1) a b' sem)
80
81  let weakenMany : type a a' b. a TEnv.hlist -> a' TEnv.hlist
82                                  -> (b, a) sem -> (b, a') sem
83    = fun a b sem ->
84      let l1 = TEnv.length a in
85      let l2 = TEnv.length b in
86      let lenDiff = l2 - l1 in
87      go lenDiff a b sem
88
89  type 'a envi = { runEnvI : 'e. 'e TEnv.hlist -> ('a, 'e) sem}
90
91  let runOpen : type a b.(a envi -> (b envi)) -> (b, a * unit) sem
92    = fun f ->
93     let gA = TEnv.HCons ((), TEnv.HNil) in
94      let x  = { runEnvI = fun g' -> weakenMany gA g' var } in
95      (f x).runEnvI gA
96
97  module HListEnvI = HList(struct type 'a t = 'a envi end)
98
99  type ('xs, 'x) sig2 = |
100
101 type (_,_) SemRep =
102 | TR : ('xs,'env,'r) wapp * ('x, 'r) sem
103             -> ('env, ('xs, 'x) sig2) SemRep
104
105
106 type (_) HoasRep =
107 | UR : 'xs TEnv.hlist * ('xs HListEnvI.hlist -> 'x envi)
108                                       -> ('xs,'x) sig2 HoasRep
109
110 module HoasRepEnv = HList(struct type 'a t = 'a HoasRep end)
111
112 module SemRepEnv(E: TypeM) =
113               HList(struct type 'a t = (E.t,'a) SemRep end)
114
115 type ('env,_) SemRepEnv_hlist =
116 | THNil  : ('env,unit) SemRepEnv_hlist
117 | THCons : ('env,'s) SemRep * ('env, 'ss) SemRepEnv_hlist
118                            -> ('env,'s * 'ss) SemRepEnv_hlist
119
120 module MapToSemRepHList(H : HLIST) = struct
121   type 'env map_func = { f : 'a. 'a H.el -> ('env,'a) SemRep }
```

```
122     let rec map : type r. 'env map_func -> r H.hlist
123                           -> ('env,r) SemRepEnv_hlist
124       = fun func l ->
125         match l with
126         | H.HNil -> THNil
127         | H.HCons(x,r) -> THCons(func.f x, map func r )
128   end
129
130   type ('ss,'rr) semTerm =
131     {f: 'env. ('env,'ss) SemRepEnv_hlist -> ('rr, 'env) sem }
132
133   let liftCore :
134     type ss rr. (ss,rr) semTerm -> ss HoasRepEnv.hlist -> rr envi
135     = fun ff ks ->
136       { runEnvI = fun (type env) (e : env TEnv.hlist) ->
137         let module App = HListP(TEnv) in
138         let module M_map = MapToSemRepHList(HoasRepEnv) in
139         let rec mkXs : type env ys ys_env. env TEnv.hlist
140             -> ys TEnv.hlist
141             -> (ys, env, ys_env) wapp
142             -> ys_env TEnv.hlist -> ys HListEnvI.hlist
143           = fun p ys wit te ->
144               match ys, wit, te with
145               | HNil, _, _ -> HNil
146               | (HCons(_,ys')),AppStep(wit'),HCons(_,te') ->
147                 let x = { runEnvI = fun e' ->
148                         weakenMany te e' H.var} in
149                 HCons(x,mkXs p ys' wit' te')
150           in
151         let cnv : type env xs x. env TEnv.hlist -> xs TEnv.hlist
152                     -> (xs HListEnvI.hlist -> x envi)
153                     -> (env, (xs,x) sig2) SemRep = fun e e1 k ->
154           match App.append_hlist e1 e with
155           | AppHList(wit, ex_e) ->
156             let xs = mkXs e e1 wit ex_e in
157             TR(wit, (k xs).runEnvI ex_e) in
158         let conv : type env s. env TEnv.hlist -> s HoasRep
159                     -> (env,s) SemRep
160           = fun e ur -> match ur with
161             | UR(e1, k) -> cnv e e1 k in
162         ff.f (M_map.map {f = fun xs -> conv e xs} ks)
163       }
164
165   module CurryUncurry(H : HLIST) = struct
166
```

```
167     type (_,_,_) cspec =
168     | Z : ('r, 'r, unit) cspec
169     | S : ('r, 'f, 'xs) cspec
170           -> ('r, 'a H.el -> 'f, ('a * 'xs)) cspec
171
172     let rec uncurry : type r f xs. (r,f,xs) cspec -> f
173                                          -> xs H.hlist -> r
174       = fun s h args -> match s, args with
175         | Z ,   _            -> h
176         | S ss, HCons(x,xs) -> uncurry ss (h x) xs
177
178     let rec cspec2TEnv : type r f xs. (r, f, xs) cspec
179                                          -> xs TEnv.hlist
180       = fun n -> match n with
181         | Z -> TEnv.HNil
182         | S s' -> TEnv.HCons ((), cspec2TEnv s')
183   end
184
185   module CU_EnvI = CurryUncurry(HListEnvI)
186
187   let toHoasRep :
188     type f xs x.
189     (x envi, f, xs) CU_EnvI.cspec
190     -> f
191     -> (xs,x) sig2 HoasRep
192     = fun cs f -> UR (CU_EnvI.cspec2TEnv cs, CU_EnvI.uncurry cs f)
193
194   type (_,_,_) tf_spec =
195     | SNil  : ('rr, 'rr, unit) tf_spec
196     | SCons : ('x envi, 'f, 'xs) CU_EnvI.cspec *
197                       ('rr, 'ff, 'ss) tf_spec ->
198                 ('rr, 'f -> 'ff, (('xs, 'x) sig2 * 'ss)) tf_spec
199
200   let rec toF :
201     type rr ff ss.
202     (rr envi,ff,ss) tf_spec -> (ss HoasRepEnv.hlist -> rr envi) -> ff
203     = fun tfs h -> match tfs with
204         | SNil -> h HNil
205         | SCons(cs,tfs) -> fun k
206                -> toF tfs (fun r -> h (HCons(toHoasRep cs k,r)))
207
208   type (_, _,_,_,_) ff_spec =
209     | FNil  : ('dummy, 'env,'rr, ('rr,'env) sem,unit) ff_spec
210     | FCons : ('xs,'env,'app) wapp  *
211               ('dummy,'env,'rr,'ff,'ss) ff_spec
```

```
212               -> ('app * 'dummy, 'env, 'rr, ('x,'app) sem
213                   -> 'ff , (('xs,'x) sig2 * 'ss)) ff_spec
214
215   type (_,_,_,_,_) wapp_cspec =
216     | XZ : (unit, 'app, 'app, 'r, 'r) wapp_cspec
217     | XS : ('xs,  'ys,  'app, 'r, 'f) wapp_cspec
218             -> (('x * 'xs), 'ys, ('x * 'app),
219                              'r, 'x envi -> 'f) wapp_cspec
220
221   let rec to_wapp :
222     type xs ys app r f. (xs, ys, app, r, f) wapp_cspec ->
223                                             (xs,ys,app) wapp
224     = function
225       | XZ   -> AppNil
226       | XS n -> AppStep (to_wapp n)
227
228   let rec to_cspec :
229     type xs ys app r f. (xs, ys, app, r, f) wapp_cspec
230                               -> (r, f, xs) CU_EnvI.cspe
231     = function
232       | XZ  -> Z
233       | XS n -> S (to_cspec n)
234
235   type (_,_,_,_,_,_) arg_spec =
236     | XNil  : ('dummy, 'env, 'rr, 'rr envi,
237                        ('rr,'env) sem, unit) arg_spec
238     | XCons : ('xs, 'env, 'app, 'x envi, 'f) wapp_cspec *
239               ('dummy, 'env, 'rr, 'toFunc,
240                              'fromFunc, 'ss) arg_spec ->
241               ('app * 'dummy, 'env, 'rr, 'f -> 'toFunc,
242               ('x,'app) sem -> 'fromFunc,
243               (('xs,'x) sig2 * 'ss)) arg_spec
244
245   let rec to_tf_spec :
246     type dummy env rr toFunc fromFunc ss.
247     (dummy, env, rr, toFunc, fromFunc, ss) arg_spec ->
248     (rr envi, toFunc, ss) tf_spec
249     = function
250       | XNil -> SNil
251       | XCons(x, xs) -> SCons(to_cspec x, to_tf_spec xs)
252
253   let rec to_ff_spec :
254     type dummy env rr toFunc fromFunc ss.
255     (dummy, env, rr, toFunc, fromFunc, ss) arg_spec ->
256     (dummy, env, rr, fromFunc, ss) ff_spec
```

```
257     = function
258       | XNil -> FNil
259       | XCons(x,xs) -> FCons(to_wapp x, to_ff_spec xs)
260
261   let toF :
262     type dummy env rr toFunc fromFunc ss.
263     (dummy, env, rr, toFunc, fromFunc, ss) arg_spec ->
264     (ss HoasRepEnv.hlist -> rr envi) -> toFuc
265     = fun s -> toF (to_tf_spec s)
266
267   let rec fromFunc :
268     type dummy env res tt ss. (dummy, env, res, tt, ss) ff_spec ->
269                  tt -> ((env, ss) SemRepEnv_hlist -> (res,env) sem)
270     = function
271         | FNil -> fun x _ -> x
272         | FCons (w1, fs) -> fun f -> function
273             | THCons(TR(w2,t),ts) ->
274                 match wapp_functional w1 w2 with
275                 | Refl -> fromFunc fs (f t) ts
276
277   let fromF :
278     type dummy env rr toFunc fromFunc ss.
279     (dummy, env, rr, toFunc, fromFunc, ss) arg_spec ->
280     fromFunc -> ((env, ss) SemRepEnv_hlist -> (rr,env) sem)
281     = fun s -> fromFunc (to_ff_spec s)
282
283 end
```

ソースコード 1　OCaml 版 EbU の実装

```
1  module STLC = struct
2    module VEnv = HList(struct type 'a t = 'a end)
3
4    type ('a, 'e) stlc = { runSim : 'e VEnv.hlist -> 'a }
5
6    module VarSTLC = Variables (
7      struct
8        type ('a, 'b) sem = ('a, 'b) stlc
9
10       let var = { runSim = function VEnv.HCons (x, _) -> x }
11
12       let weaken = fun stlc ->
13         { runSim = function VEnv.HCons (_, venv') ->
14                                          stlc.runSim venv' }
15   end
16   )
```

```
17 end
18
19 module SemSTLC = struct
20   open STLC
21
22   let appSem = fun fTerm aTerm ->
23     { runSim = fun venv ->
24       let f = fTerm.runSim venv in
25       let x = aTerm.runSim venv in
26       f x }
27
28   let lamSem = fun bTerm ->
29     { runSim = fun venv x ->
30       let g' = VEnv.HCons (x, venv) in
31       bTerm.runSim g' }
32 end
33
34 module type HSTLC = sig
35   type 'a expr
36   val app : ('a -> 'b) expr -> 'a expr -> 'b expr
37   val lam : ('a expr -> 'b expr) -> ('a -> 'b) expr
38   val runOpenSTLC : ('a expr -> 'b expr) -> ('b, 'a * unit) stlc
39 end
40
41 module STLC_sem : HSTLC with type 'a expr = 'a STLC.VarSTLC.envi =
42 struct
43   type 'a expr = 'a VarSTLC.envi
44   open STLC open VarSTLC
45
46   let app :
47     type a b. (a -> b) envi -> a envi -> b envi = fun e1 e2 ->
48     let argSpec = XCons (XZ, XCons (XZ, XNil)) in
49       toF argSpec
50         (liftCore' {f = fun x -> fromF argSpec SemSTLC.appSem x}) e1 e2
51
52   let lam :
53     type a b. (a envi -> b envi) -> (a -> b) envi = fun e ->
54     let argSpec = XCons (XS XZ, XNil) in
55       toF argSpec
56         (liftCore' {f = fun x -> fromF argSpec SemSTLC.lamSem x}) e
57
58   let runOpenSTLC :
59     type a b.(a expr -> b expr) -> (b, a * unit) stlc = fun f ->
60     VarSTLC.runOpen f
61 end
```

```
62
63 let runSTLC = fun t x ->
64   let g = VEnv.HCons (x, VEnv.HNil) in
65   t.runSim g
```

ソースコード 2　STLC の埋め込み

```
1   module ILC = struct
2   module FromFst = struct
3     type _ t =
4     | FFst : 'a -> ('a * 'd) t
5   end
6
7   module FromSnd = struct
8     type _ t =
9     | FSnd : 'd -> ('a * 'd) t
10  end
11
12
13  module H = HList(FromFst)
14  module D = HList(FromSnd)
15
16  open FromSnd
17  open FromFst
18  open H
19  open D
20
21  type ('x, 'xs) ilc = ('xs H.hlist -> 'x FromFst.t) *
22            ('xs H.hlist -> 'xs D.hlist -> 'x FromSnd.t)
23
24  let varN :
25    type x xs. (x * xs) H.hlist -> x FromFst.t =
26    function (HCons(FFst x,_)) -> FFst x
27
28  let varD :
29    type x xs. (x * xs) H.hlist -> (x * xs) D.hlist ->
30                                              x FromSnd.t
31    = fun _ -> function (HCons(FSnd d,_)) -> FSnd d
32
33  module VarILC = Variables (
34    struct
35      type ('x, 'xs) sem = ('x, 'xs) ilc
36      let var : type x xs. (x, (x * xs)) ilc = (varN, varD)
37      let weaken : type x y xs. (x, xs) ilc -> (x, (y * xs)) ilc
38        = fun (f,df) ->
39        (function (HCons(_,env)) -> f env),
```

```
40          (function (HCons(_,env)) -> function (HCons(_,denv))
41                                                  -> df env denv)
42      end
43    )
44  end
45
46  module SemILC = struct
47    open ILC
48
49    let addSem :
50      type env. (int * int, env) ilc -> (int * int, env) ilc
51                                        -> (int * int, env) ilc
52      = fun (f, df) (g, dg) ->
53        (fun theta -> let FFst v = f theta in let FFst w = g theta in
54        FFst (v + w)) ,
55        (fun theta dtheta -> let FSnd dv = df theta dtheta in
56        let FSnd dw = dg theta dtheta in FSnd (dv + dw))
57
58    let mulSem :
59      type env. (int * int, env) ilc -> (int * int, env) ilc
60                                        -> (int * int, env) ilc
61        = fun (f, df) (g, dg) ->
62            (fun theta ->
63              let FFst v = f theta in
64              let FFst w = g theta in
65              FFst (v * w)),
66            (fun theta dtheta ->
67              let FFst v = f theta in
68              let FFst w = g theta in
69              let FSnd dv = df theta dtheta in
70              let FSnd dw = dg theta dtheta in
71              FSnd (v * dw + dv * w + dv * dw)
72              )
73
74    let fstSem :
75      type env a da b db. ((a * b) * (da * db) , env) ilc ->
76                                              (a * da, env) ilc
77          = fun (f, df) ->
78            (fun theta ->
79              let FFst (v, _) = f theta in
80                FFst v),
81            (fun theta dtheta ->
82              let FSnd (dv,_) = df theta dtheta in
83                FSnd dv
84              )
```

```
85
86    let sndSem :
87      type env a da b db. ((a * b) * (da * db) , env) ilc ->
88                                                  (b * db, env) ilc
89          = fun (f, df) ->
90            (fun theta ->
91              let FFst (_, v) = f theta in
92                FFst v),
93            (fun theta dtheta ->
94              let FSnd (_, dv) = df theta dtheta in
95                FSnd dv
96              )
97
98    let letSem :
99      type env a b. (a, env) ilc -> (b, (a * env)) ilc ->
100                                                    (b, env) ilc
101         = fun (f, df) (g, dg) ->
102             (fun theta ->
103               let FFst v = f theta in
104               let FFst w = g (HCons(FFst v,theta)) in
105                 FFst w),
106             (fun theta dtheta ->
107               let FFst v = f theta in
108               let FSnd dv = df theta dtheta in
109                 dg (HCons(FFst v, theta)) (HCons(FSnd dv,dtheta))
110               )
111 end
112
113 module type HILC = sig
114   type 'a expr
115
116   val add : (int * int) expr -> (int * int) expr -> (int * int) expr
117   val mul : (int * int) expr -> (int * int) expr -> (int * int) expr
118   val ( let* ) : 'a expr -> ('a expr -> 'b expr) -> 'b expr
119   val fst : (('a * 'b) * ('da * 'db)) expr -> ('a * 'da) expr
120   val snd : (('a * 'b) * ('da * 'db)) expr -> ('b * 'db) expr
121
122
123   val runOpen : (('a * 'da) expr -> ('b * 'db) expr) ->
124                                      ('a -> 'b) * ('a -> 'da -> 'db)
125
126 end
127
128 module HILC_sem : HILC  with type 'a expr = 'a ILC.VarILC.envi =
129 struct
```

```
130   open ILC open VarILC open SemILC
131   type 'a expr = 'a envi
132
133   let add = fun e1 e2 ->
134     let argSpec = XCons (XZ, XCons (XZ, XNil)) in
135       toF argSpec
136         (liftCore {f = fun x -> fromF argSpec addSem x})
137           e1 e2
138
139   let mul = fun e1 e2 ->
140     let argSpec = XCons (XZ, XCons (XZ, XNil)) in
141       toF argSpec
142         (liftCore {f = fun x -> fromF argSpec mulSem x}) e1 e2
143
144   let ( let* ) = fun e1 e2 ->
145     let argSpec = XCons (XZ, XCons (XS XZ, XNil)) in
146       toF argSpec
147         (liftCore {f = fun x -> fromF argSpec letSem x}) e1 e2
148
149   let fst = fun e ->
150     let argSpec = XCons (XZ, XNil) in
151       toF argSpec
152         (liftCore {f = fun x -> fromF argSpec fstSem x}) e
153
154   let fst = fun e ->
155     let argSpec = XCons (XZ, XNil) in
156       toF argSpec
157         (liftCore {f = fun x -> fromF argSpec SndSem x}) e
158
159   let runOpen :
160     type a da b db. ((a * da) expr -> (b * db) expr) ->
161                                 (a -> b) * (a -> da -> db)
162     = fun f ->
163       let (g, dg) = VarILC.runOpen f
164       in (fun x -> let FFst v = g (HCons(FFst x, HNil)) in v),
165          (fun x dx ->
166           let FSnd dv =
167             dg (HCons (FFst x,HNil)) (HCons (FSnd dx,HNil)) in dv)
168 end
```

ソースコード 3　ILC の埋め込み